Addressing Variability in a Guidance, Navigation, and Control Flight Software Product Line

David McComas[1], Stephen Leake[1], Michael Stark[2],
Maurizio Morisio[2], Guilherme Travassos[2], and Michael White[3],

[1] NASA Goddard Space Flight Center,
Greenbelt, Maryland USA
{David.C.McComas.1,
Stephen.A.Leake.1}@gsfc.nasa.gov,
[2] Software Engineering Lab,
Michael.E.Stark.1@gsfc.nasa.gov
morisio@polito.it
ght@cos.ufrj.br
[3] Johns Hopkins University Applied Physics Lab,
Laurel, Maryland USA
Michael.J.White@jhuapl.edu

Abstract. The NASA Goddard Space Flight Center is developing a guidance, navigation, and control flight software product line that includes both processes and their accompanying products. The processes include a domain and application engineering process that have been influenced by Synthesis[1] and FAST[2]. The products include graphical and textual analysis/design documents and the flight software repository. We are using UML (Unified Modeling Language) stereotypes to represent variability in our domain analysis models. This paper focuses on how variability is addressed during each phase of the domain and application engineering processes. The techniques are illustrated using the Celestial Body subdomain. The GNC FSW product line is a work in progress so many concepts presented in this paper have not fully matured.

1 Introduction

The NASA Goddard Space Flight Center Flight Software Branch (FSB) is developing a Guidance, Navigation, and Control (GNC) Flight Software (FSW) product line. The demand for increasingly more complex flight software in less time while maintaining the same level of quality has motivated us to look for better FSW development strategies. The GNC FSW product line will address the core GNC FSW functionality that has been very similar on many recent low/near Earth missions in the last 10 years. Unfor-

tunately these missions have not realized significant drops in development cost since a systematic approach towards reuse has not been adopted. In addition, new demands are continually being placed upon the FSW which mean the FSB must become more adept at providing the core GNC FSW functionality so it can accommodate the additional requirements. The GNC FSW product is being developed to address these issues.

For our purposes, domain engineering includes the engineering activities needed to produce reusable artifacts for a domain. Application engineering refers to developing an application in the domain starting from reusable artifacts. Domain engineering is the foundation for emerging product line software development approaches [3]. A product line is "A family of products designed to take advantage of their common aspects and predicted variabilities" [2].

The focus of this paper is on how the GNC FSW product line manages variability. Existing domain engineering approaches do not enforce any specific notation for domain analysis or commonality and variability analysis. Usually, natural language text is the preferred tool. The advantage is the flexibility and adaptability of natural language. However, one has to be ready to accept its well-known drawbacks, such as ambiguity, inconsistency, and contradictions.

While most domain analysis approaches are functionally oriented, the idea of applying the object-oriented approach in domain analysis is not new [3]. Several authors [4, 5] propose to use UML [7] as the notation underlying domain analysis. Keepence and Mannion [6] propose to use the design pattern notation and style to represent discriminants, or features that distinguish one application from another. Our work is based on the same idea of merging UML and domain analysis. Further, we propose a few extensions to UML in order to express variability, and we define precisely their semantics so that a tool can support them.

The paper is organized as follows. Section 2 outlines the product line processes and identifies where variability must be addressed. Section 3 describes the product line products with respect to how they accommodate variability. The Celestial Body subdomain is used as a working example. Section 4 summarizes our results to date and describes what we plan to do in the future.

2 Process Overview

We initially tried to adhere to the Synthesis [1] process, but our reference material lacked the depth we needed to actually implement a product line. In addition Synthesis introduces a new set of terminology that may hinder the acceptance of the product line when we introduced it to the rest of the FSB. Since we know how to develop individual applications, we extended the process to apply to developing applications within the context of a product line using reusable assets. We defined our own product line domain and application processes (see Figure 1) by decomposing the processes into the familiar analysis, design, implementation, and verification lifecycle phases.

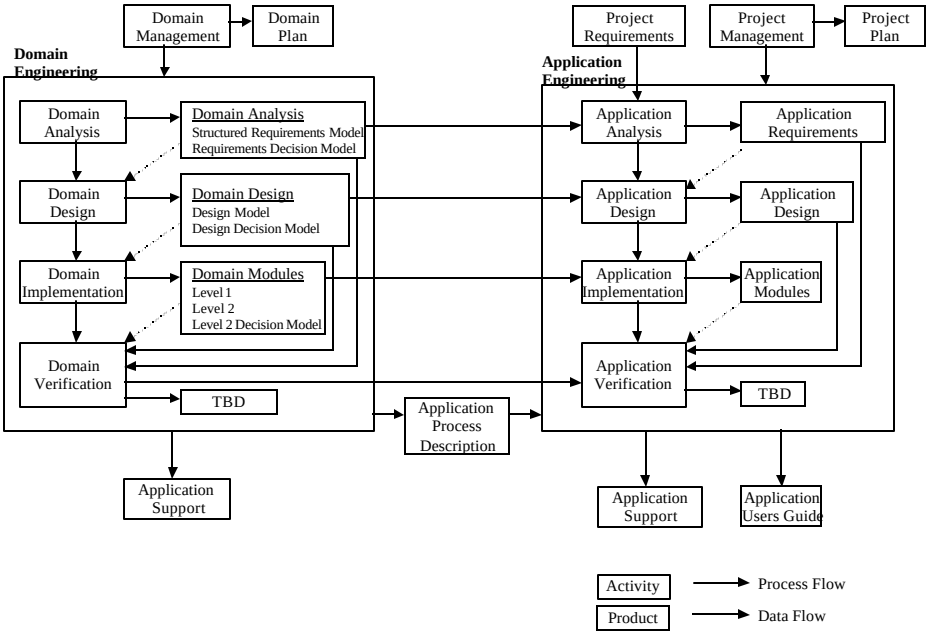

Figure 1 - GNC FSW Product Line Processes

Initially we identified *what* products are associated with each process, but we didn't know *how* information would be represented in the products. To solve this problem we adopted a strategy to develop a series of increasingly more complex prototype applications. Application complexity is a function of the difficulty of the domain problem being solved and the runtime environment. For example, our first application is considered simple because it does not interface to external hardware and we are developing it independent of the embedded flight software context (i.e we can verify the application on a PC without a simulator). As we work through each application within the context of a product line we will discover the information that would need to be represented in order to be useful. Based on experience we had identified several subdomains within the GNC FSW domain to serve as our reference architecture. Our intention is to include each subdomain in at least one prototype application.

Domain engineering and application engineering govern the GNC FSW product line. Domain engineering produces engineering products that describe a domain while application engineering produces an application starting from and reusing domain engineering products. Domain engineering is composed of four activities. Domain analysis defines the scope of the domain, describes the domain and performs commonality and variability analysis on it. Its main deliverable is a document in UML and text. Domain design focuses on design decisions for the domain and delivers a document in UML and text. Domain implementation develops source code modules. Finally domain verification provides unit test for verifying individual modules and a testing framework to be used by the application verification process. Each domain engineering product must represent commonality and variability in a manner that is usable by ap-

plication engineering. This can only be achieved by bounding the domain problem space so the variability is predictable.

Application engineering is composed of the same four phases, analysis, design, implementation, and verification. These phases have the usual meaning as for current projects; however they do not develop the corresponding deliverables from scratch, but simply modify and adapt the corresponding domain engineering deliverable. Heuristics guide this process. They are collected and documented during domain engineering. The application engineer uses them to make analysis, design and implementation decisions.

3 Products and Variability

This section describes how variability is captured and used in each of the domain and application engineering processes. At the time of this writing we are in the process of developing our second prototype application so many concepts have not been fully matured.

These sections use the GNC FSW Celestial Body subdomain as an example to help clarify concepts. A celestial body is any object in space within our solar system that is within scope of the GNC FSW product line domain. This includes the Earth, the Sun, the Moon, spacecraft, comets, and asteroids. Stars other than the Sun are not characterized as celestial bodies because the required information about distant stars is minimal.

3.1 Analysis

This section presents the graphical notation used in the domain analysis phase to capture the domain variability and describes the application analysis process used to resolve the variability. Figure 2 shows the Celestial Body subdomain analysis deliverable.

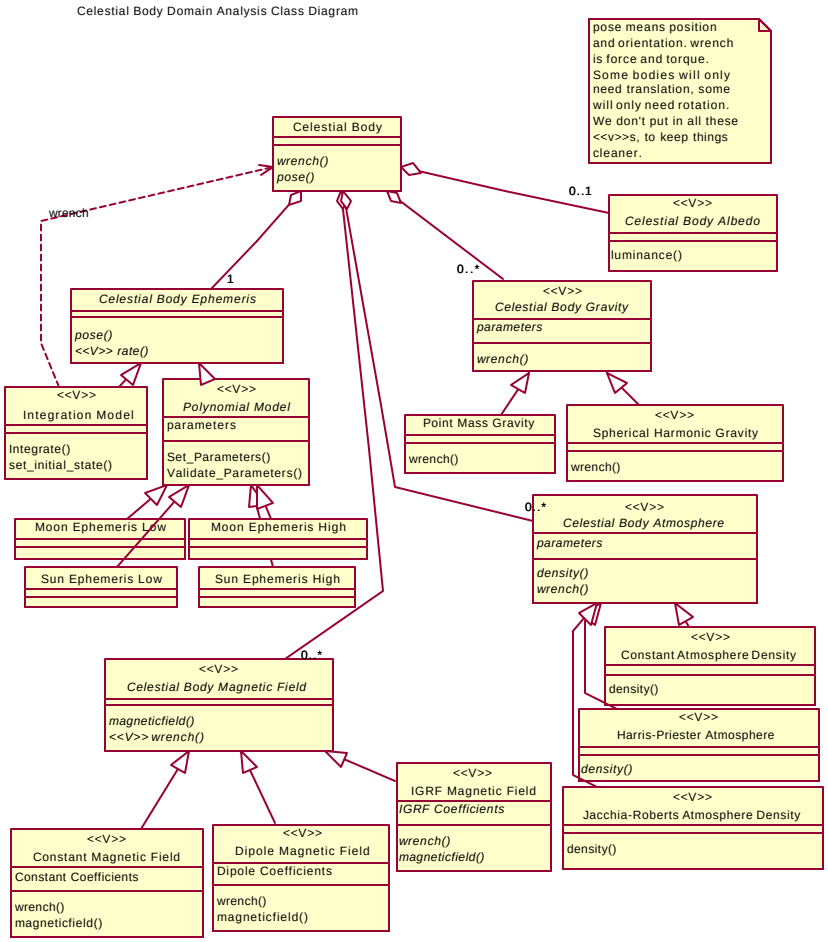

Figure 2 Celestial Body Domain Analysis Diagram

The notation is based on UML as applied to object oriented (OO) analysis. In traditional OO analysis it is assumed that:

- A single application's requirements are analyzed to capture its concepts (classes), properties (attributes), behaviors (operations) and static relationships.
- All elements in the OO analysis model (classes, attributes, operations, relationships) are part of the application.

In domain analysis:

- A domain (i.e. a set of applications) is analyzed using knowledge from past systems and anticipated requirements of future systems.
- Only a subset of the analysis model is used for a particular application.

Elements in the domain analysis are part of the domain, but not always part of the application. In other words commonality, variability and exclusion need to be represented. Our approach is to use a special symbol <<V>> to represent variability, and to interpret elements that are not tagged by a <<V>> as a commonality. Exclusion is not represented in the diagrams (however, elements excluded and rationales for exclusion are detailed in the text that accompanies the UML).

Figure 2 shows that all *Celestial Bodies* contain a *Celestial Body Ephemeris* (an ephemeris defines the position and optionally the velocity of a body at a specific time). In other words, *Celestial Body* and *Celestial Body Ephemeris* are common to all applications. The class celestial body gravity *Celestial Body Gravity* is tagged by a <<V>>, which means it could be present in some applications while not in others.

The same variability concept applies to operations, attributes, and arguments of operations. Operation *pose()* in class *Celestial Body Ephemeris* is present in all applications, while in the same class *rate()* may or may not be present.

Generalization and aggregation relationships can change, indirectly, the availability of a class in an application. Consider classes *Point Mass Gravity* and *Spherical Harmonic Gravity*. Since they are a specialization of the variable class *Celestial Body Gravity* they can only be part of an application if *Celestial Body Gravity* has been selected. The same applies to aggregation, or part-of, relationships. If class A is part-of class B, and B is variable, A is in the application only if B is in the application. Since generalization and aggregation relationships define sets of classes (the set of specialization and the set of parts, respectively) our notation supports the concept of discriminants (differentiating system features) [6]. A complete description of our UML extensions can be found in [8].

Application Analysis

Application analysis begins with the selection of the models to include in the application. The variability shown in Figure 2 must be resolved. Note that the designer of a mission's orbit is generally not a software engineer or UML expert, so the variability needs to map to mission concepts in a manner that is comprehensible by the mission analysts. We are currently investigating whether UML and text are the best way to represent this information for the analysts.

To illustrate the application analysis process, we will consider two types of missions from the orbit mechanics perspective: one that orbits a Lagrange point, and one that is in a low Earth orbit. We will not define these terms in detail, the significant thing for the application engineer is the knowledge that Lagrange points are far enough away from Earth that Earth's atmosphere and magnetic field have no significant effect on the orbit. Figures 3 and 4 show the class diagrams for the low Earth orbit (LEO) and the Lagrange orbit, respectively.

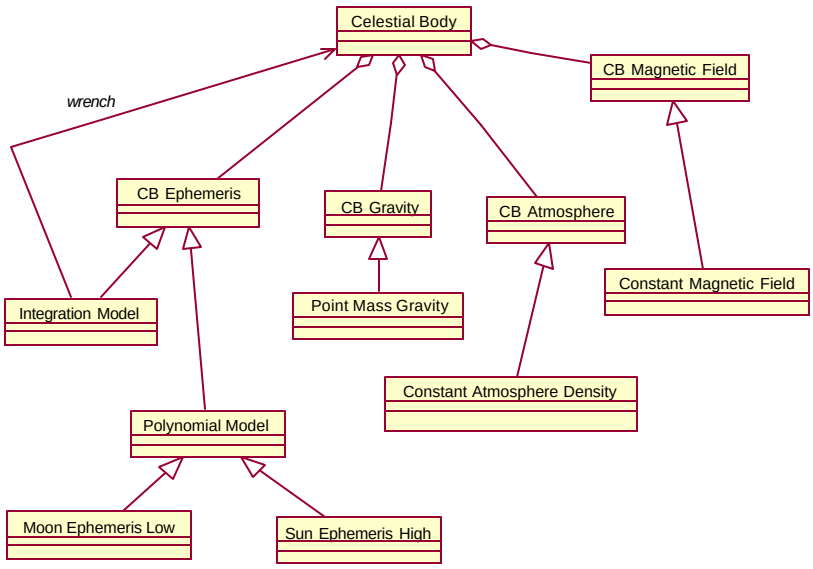

Figure 3 - Low Earth Orbit Application Class Diagram

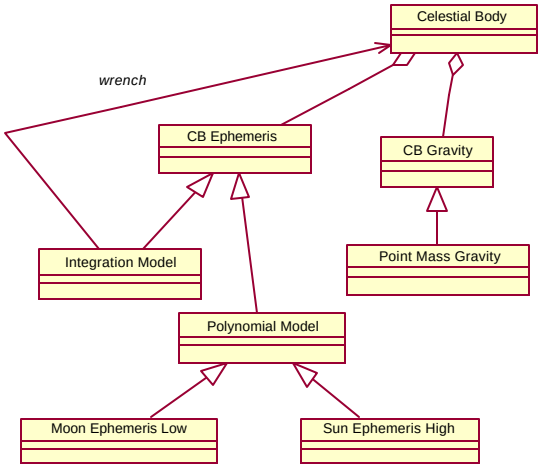

Figure 4 - Lagrange Orbit Application Class Diagram

The obvious difference between the two class diagrams is the inclusion of atmosphere and magnetic field modeling for the low Earth orbit mission. These diagrams also implicitly illustrate some rules for variability and generalization. If a superclass has a class-level variability and is not included in the application, none of its subclasses may appear; for example excluding atmosphere modeling from the Lagrange mission automatically excludes the three concrete subclasses that implement the models. Including a superclass does not automatically include all subclasses, however. In the low Earth orbit mission, both the atmosphere and magnetic field models include only one subclass out of the 3 possibilities for each.

These rules are fairly simple, and are documented in [8]. They can become both more complex and more domain dependent when one considers the relationships between objects of two given classes. For example, Figure 2 shows that the *Celestial Body* class always contains exactly one *Celestial Body Ephemeris*, which is the model that propagates an orbit. What it does not show is that to model the Sun as a *Celestial Body* object, this object should not contain instances from either of the Moon Ephemeris subclasses. This type of constraint could be enforced by some form of domain rule base, or by application inspection checklists that include usage rules. It is not yet clear which will be a more useful approach for the flight software problem.

Another issue that needs to be resolved is exactly what UML diagrams are useful for product line analysis tasks, and what diagrams could be useful as generated products for the application. During *Celestial Body* domain analysis, we also generated an object diagram showing *Celestial Body* objects for a spacecraft, the Sun, the Earth, and the Moon, and a corresponding interaction diagram showing how the spacecraft uses an *Integration Model* to propagate orbit, modeling the effects of the Sun, Earth and Moon. These added diagrams do not show any variability, but may be useful in helping an application engineer decide how to resolve variability in the domain analysis model.

Similarly, we examined the use of object diagrams as part of application analysis. The motivation was to clarify that there are distinct steps for choosing models by resolving the variabilities, creating objects, and defining the relationships between the objects. While this paper has focused on the issue of variabilities and how they are resolved, our implementation prototyping has shown the need for defining objects, their relationships, and default initial parameters for the application system. The results of these prototypes need to be fed back to the analysis and design stages of the application engineering process.

The result was that even an object diagram generated from the simpler Lagrange application class diagram seemed cluttered, indicating potential problems when the domain model is populated with more classes. It is still important to distinguish these two steps in the application engineering process, but there may be better ways to represent each one. At this point the prototype team has not established what diagrams or text documents (if any) are needed for application engineering. Creating candidate diagrams during the prototyping process will help clarify this issue, as well as understanding the requirements for tools supporting application engineers.

3.2 Design and Implementation

Design provides a solution to the problem defined by the analysis process. Our design must accommodate the variability expressed in the analysis model within the context of the GNC FSW environment. The GNC FSW environment has several constraining features. The GNC FSW resides on a remote embedded system so there is no direct user interface. User interaction is achieved via commands, telemetry, events, and tables. Commands direct the GNC FSW to perform an action. Commands can originate from the ground or from other onboard components. Telemetry is data that is exported by the GNC FSW. Events are time-stamped notifications of a particular GNC FSW state. Telemetry and events can be received by the ground and by other onboard components. Tables are logical groups of parameters that can be loaded/dumped to/from the spacecraft as a complete set. Additional GNC FSW constraints typically include hard real-time performance requirements, customized hardware, out-of-date processors, limited memory, and restrictions on the use of dynamic memory facilities.

The domain design must provide variability mechanisms for classes, for groups of classes (components), and between components. As we work through our prototype applications we are documenting our design decisions and rules. We expect to have a consistent set of architectural elements, design patterns, and rules for using them in order to implement a complete and consistent domain design.

As an example, consider how the *Celestial Body* class in Figure 2 is an aggregate of several classes whose inclusion is variable. A technique for addressing this situation is to define an *Add()* operation to the *Celestial Body* class for each <<v>> class it contains. If an application requires the <<v>> class then the following steps must be taken:

1. Create a static instance of each *Celestial Body* aggregate object. Note dynamic memory allocation is typically not used for FSW.
2. During initialization, the component that contains a *Celestial Body* would add the aggregates objects that are required for the specific application using the *Add()* operation.

We are currently in the process of defining architectural elements that will support commands, telemetry, events, and tables while providing the flexibility to allow the components to be assembled in multiple configurations. Standardized interfaces, parameterized components, and hierarchical component structures are some of the techniques we are investigating.

Application design must utilize the domain design mechanisms to implement the application analysis model. In the "add" example above, the addition of the Add() operation and the application development steps are part of the domain design. Actually carrying out the steps is application design. In general application design must identify the required components/objects (this is derived from the application analysis model), define parameters, and resolve associations (object dependencies). Currently, we have not felt the need to generate any additional design information other than the

code itself. This is due in part to good coding standards. It also alleviates the need to maintain consistency between an application design document and the code

Domain implementation consists of coding the domain analysis models using domain design elements, patterns, and rules. Common domain classes are simply coded. We are still exploring various techniques for handling implementation variability. These techniques include using makefiles and preprocessor directives for conditional code inclusion. At the class operation level an exception could be raised if an application tries to use a function that has not been instantiated.

3.4 Verification

Our current prototypes have not been verified, so this section reflects our plans. Domain verification ensures the correctness, consistency, and completeness of the domain engineering work products [1]. The techniques employed will resemble our current techniques for verifying a single application. These include inspection and review of analysis, design, code and test products. Following code inspection, unit testing is performed. A unit is defined as a single interface file, together with the body files that implement the unit. In C/C++, the interface is a header file. In Ada an interface is a package specification. Components are groups of tightly coupled units. Component testing will also be performed. Both unit and component tests will be configuration managed with the source code.

In our *Celestial Body* example, a unit test is developed for each class. For abstract classes, test-specific concrete classes will be developed to fully exercise the abstract interface. It is not feasible to test all permutations of the variability expressed in the class diagrams. However, a representative test suite may be useful. For example, at the component level, a low Earth orbit and a Lagrange orbit test could be defined. These would test the two most common *Celestial Body* configurations. We must keep in mind that random application specifications will not be generated. A domain expert will be part of the application specification process and to a domain expert, the variability resolution for a subdomain such as *Celestial Body* is straightforward.

An important aspect of flight software verification is the execution of unit tests on an application's target environment. This requires the unit tests written during domain verification to be portable and repeatable. This is also another reason to configuration manage the unit tests with the source code.

As part of our product line development strategy we will use the product line to implement previous missions. This will verify the domain process and products within the context of a previous problem. However, application verification and validation will always be necessary, regardless of the level of domain verification. We are also considering platform-independent closed-loop testing on a simulated flight system. This would allow us to prototype applications, but again its verification potential is limited.

4 Ongoing Work

We have completed application 1 in C, C++, and Ada. We are currently implementing application 2. We made the decision to use Ada as our sole prototyping language for the remaining applications to help speed up the effort. The final prototype will be implemented in C++ as well as Ada. This will help us identify language specific issues and it will also allow us to present the conceptual GNC FSW product line to the FSB in two familiar languages.

Based on the length of each subsection within section 3 it is obvious that we have made the most progress in analysis. However, even within analysis we have not explored behavioral models and their impact on variability. Our plan is to continue working through the prototypes in an effort to mature our processes and their associated products. Once these have matured we will fully explore automating portions of the process.

References

1. Software Productivity Consortium, Reuse-Driven Software Processes Guidebook, SPC-92019-CMC version 02.00.03 November 1993.
2. Weiss, David M. and Chi Tau Robert Lai Software Product-Line Engineering: A Family-Based Software Development Approach. Addison-Wesley, 1999
3. John Foreman. *Product Line Based Software Development- Significant Results, Future Challenges*. Software Technology Conf., Salt Lake City, UT, 1996.
4. S. Gossain, D. Batory, H. Gomaa, M. Lubars, C. Pidgeon, and E. Seidewitz, Objects anddomain engineering (panel), Proceedings of OOPSLA95, 1995.
5. Cohen, L. M. Northrop, Object-Oriented Technology and Domain Analysis, Fifth International Conference of Software Reuse, June 1998.
6. B. Keepence, M. Mannion, Using Patterns to Model Variability in Product Families, IEEE Software, July 1999, pp. 102-108.
7. Rumbaugh, James, Ivar Jacobson, Grady Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1998.
8. Software Engineering Laboratory, *Product line development approach for flight software,* SEL Study Brief, July 2000.